I/O Forwarding Scalable Layer

Rob Ross, Pete Beckman, Kamil Iskra — **Argonne National Laboratory**
James Nunez, John Bent, Gary Grider — **Los Alamos National Laboratory**
Steve Poole — **Oak Ridge National Laboratory**
Lee Ward — **Sandia National Laboratories**

http://www.iofsl.org/

Summary

Current leadership class machines have a few hundred thousand processing elements. Future leadership-class machines will incorporate millions of processing elements, a substantial increase in scale over existing systems. This architectural shift results from the increase in node count and the increase in number of cores per node. This extreme number of processing elements increases the potential impact of operating system (OS) noise on the system as a whole, leading system software developers to reduce the functionality of the OS in order to minimize its impact on performance.

At the same time, storage device access rates are not increasing substantially and the access rate of the I/O system as a whole is not keeping pace with increases in raw compute power. Likewise, the commercial parallel file systems typically used to organize storage devices are still designed for competitiveness in the larger business market, and their capabilities are being stretched to address leadership computing scales. This leads to very poor performance for application workloads relative to the peak performance possible from the underlying storage hardware.

A comprehensive software solution is needed to bridge the gap between processing trends and I/O systems so that leadership-class machines can most efficiently leverage the available storage resources. This solution must address the extreme degree of parallelism on the compute side, match the "lightweight" design of operating systems on these platforms, and aggregate I/O operations in order to better manage interaction with commercial parallel file systems.

This project will design, build, and distribute a scalable, unified high-end computing I/O forwarding software layer that will be adopted and supported by DOE Office of Science and NNSA. This layer will:

- Provide function shipping at the file system interface level (without requiring middleware) that enables asynchronous coalescing and I/O without jeopardizing determinism for computation,
- Offload file system function from simple or full OS client processes to a variety of targets, from another core or hardware on the same system to an I/O node on a conventional cluster or a service node on a leadership class system,
- Reduce the number of file system operations/clients that the parallel file system sees,
- Support any/all parallel file system solutions, and
- Integrate with MPI-IO and any hardware features designed to support efficient parallel I/O.

Effective I/O forwarding is a critical parallel I/O software component for leadership computing. This project will revolutionize and standardize the community's approach to accessing storage on leadership-class machines and enable efficient I/O at petascale. Building off the ZOID and ZOIDFS prototypes from the first FASTOS effort, we will create a scalable, portable framework for I/O forwarding. We will port this framework to the IBM Blue Gene, Cray XT, Roadrunner, and Linux cluster platforms and ensure it operates on a variety of file systems, including Lustre, PVFS, and PanFS.

Our expectation is that this tool will be extremely useful to the community for both production and research purposes. To enable widespread use, we will package this I/O forwarding software as an open source framework and make it available online. We will provide mailing lists for users to ask questions and obtain help in using the system. We will encourage adoption through our interactions with Cray, IBM, the SciDAC Petascale Data Storage Institute (PDSI), and the Center for Scientific Data Management (SDM).

Technical Information

A high-level overview of the forwarding layer can be found in Fig. 1.

We defined a custom I/O forwarding protocol called ZOIDFS, which is better suitable for the task than the POSIX API. It is stateless, so, e.g., instead of open(), there is lookup() (there is no equivalent of close()). There are no file offsets or even file descriptors; opaque file handles are used instead, which can be freely exchanged between compute processes (no need to call open() separately from each process of a 100k-core job). We made our API maximally flexible, so that complex I/O operations can be completed using as few API call invocations as possible. As an example, here are the prototypes of the lookup() and read() calls:

```
int zoidfs_lookup(const zoidfs_handle_t *parent_handle,
    const char *component_name, const char *full_path,
    zoidfs_handle_t *handle);
int zoidfs_read(const zoidfs_handle_t *handle,
    int mem_count, void * mem_starts[],
    const size_t mem_sizes[], int file_count,
    const uint64_t file_starts[], uint64_t file_sizes[]);
```

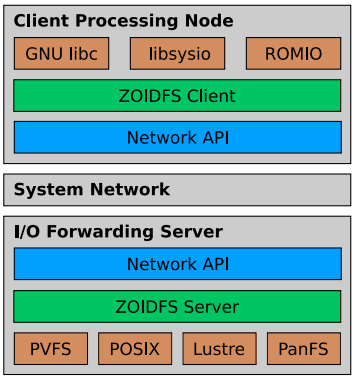

Figure 1: Software organization across client processing nodes and I/O forwarding servers.

We will build translation layers from more traditional I/O APIs such as POSIX and MPI-IO to ZOIDFS, so that all applications can benefit from our work. For POSIX, this will be done using the SYSIO library, FUSE, or a modified GNU libc, depending on the particular platform. For MPI-IO, we will add a new driver to ROMIO.

We are using BMI, a network abstraction originally developed for the PVFS filesystem, as the networking layer. BMI provides a unified asynchronous communication API on top of a variety of protocols, including TCP/IP, InfiniBand, MX, and Portals. We will add support for the Blue Gene collective network to BMI. We will also add (optional) CRC calculation to ensure the integrity of the transferred data.

The daemon running on the I/O nodes will be based in part on the tried and tested multithreaded ZeptoOS I/O Daemon (ZOID), to ensure low latency and high throughput. We will make as few intermediate data copies as possible (preferably zero for the payload data of calls such as read() and write()). Instead of indiscriminately splitting large requests into fixed-size smaller ones on the client side, we will let the I/O daemon decide on how such operations are to be performed, based on the current overall resource availability on the I/O node. This will allow us to maximize the benefits of advanced I/O optimizations, such as a cooperative cache between the I/O nodes, which is something we are collaborating on as part of another research project that we plan to leverage from.

Where available, we will use custom interfaces to the individual filesystems. E.g., PVFS offers a better performance if one calls functions from the userspace *libpvfs* directly, instead of going through the Linux kernel. We will utilize the SYSIO library as an abstraction layer for this work. Callbacks to standard POSIX API will obviously be supported as well.

Status

This project is currently in its first year of development. The networking layer is working. Forwarding of function calls, with argument marshalling and demarshalling, has been demonstrated and is currently being expanded to cover the whole ZOIDFS API. The ZOIDFS driver for ROMIO is mostly complete. A translation layer from ZOIDFS to PVFS (currently without an intermediate SYSIO layer) is available. SYSIO is complete and tested on the compute nodes of platforms such as Cray XT; an alternative API to support file handles, so that it can also be used on I/O nodes, is under development.

An alternative ZOIDFS call forwarding layer for Blue Gene, using ZOID, is complete and tested. This will in the future get merged with BMI to provide consistency across platforms.